Securing SHA-3 on Cortex M4 against DPA Attacks

Valentino Guerrini

July 3, 2023

Project Supervisor: Alessandro Barenghi

GitHub Repo:
`https://github.com/IoSonoDue2/CortexM4-ISW-Masked-SHA3`

POLITECNICO
MILANO 1863

Abstract

In this report, I document my work on a project that involved implementing the Ishai-Sahai-Wagner (ISW) protection scheme to safeguard the Secure Hash Algorithm 3 (SHA-3) against Differential Power Analysis (DPA) attacks on ARM Cortex M4 microcontrollers. I will cover the theoretical backgrounds of SHA-3, DPA attacks, and the ISW scheme, followed by a detailed methodology of my implementation. Furthermore, I have also implemented a shuffling technique, supplementing the robustness of the ISW scheme. This addition added another layer of protection, ensuring unpredictable ordering of operations, thus, making the system even more resilient against DPA attacks. Following the in-depth methodology, I present the experimental results obtained from this project.

1 Introduction

The project focuses on the implementation of the Keccak hash algorithm, also known as SHA3, on a microcontroller. The Keccak algorithm and its core, *Keccak-f1600*, are designed to withstand various types of cryptographic attacks, including collisions, pre-images, and second pre-image attacks. Despite the acknowledged security of SHA3, which makes it one of the safest cryptographic hash algorithms in the world, when implemented on a microcontroller, it is crucial to adopt protective measures to counteract DPA (Differential Power Analysis) attacks. These attacks rely on specific power variations related to the instructions executed by the microcontroller to gain information on ongoing cryptographic operations.

The project plans the implementation of the ISW (Ishai-Sahai-Wagner) masking scheme, so named after its creators. Moreover, in ths project ISW scheme has been implemented with the addition of a technique called shuffling, which alters the order of operations within an algorithm. More precisely, the order is randomized, making it difficult for an attacker to correlate power consumption observations with the specific operations of the algorithm.

The integration of ISW and shuffling greatly complicates the possibility of a side-channel attack by an aggressor. However, as discussed in Section 7, the implementation of both of these techniques can be complex and introduce additional costs in terms of computing power and memory.

2 SHA-3 Cryptographic Algorithm

In the following section, we turn our attention to the cornerstone of this project - the Secure Hash Algorithm 3. Emerging as a successor to SHA-2, this cryptographic hash function has rapidly gained recognition for its superior efficiency and robust security features. Contrary to SHA-1 and SHA-2, Keccak, also known as SHA-3, discards the Merkle-Damgård construction[1] in favor of a distinct framework known as the sponge construction. This process starts with pre-processing, wherein the input message undergoes partitioning into blocks and padding is added as needed. The sponge construction is then composed of two distinct phases:

- **The Absorbing (or input) phase** - During this phase, the message blocks, denoted by x_i, are introduced into the algorithm and processed accordingly.
- **The Squeezing (or output) phase** - In this final step, an output is generated. Notably, the length of this output is configurable, adding another layer of flexibility to this robust hashing function.

In the Keccak sponge construction, several key parameters play an integral role:

- **Bitrate (r):** The part of the state that is "exposed" and interacts with the input data.
- **Capacity (c):** The part of the state that is kept "hidden" from direct interaction with the input or output, providing security.
- **State (b):** The combination of the bitrate and capacity ($b = r + c$), which holds the function's internal state.
- **Padding (P):** Used to make the input data fit properly into the required block size.
- **Permutation function (Keccak f):** Applied to the entire state, ensuring thorough mixing of the data.
- **Output length (d):** The desired length of the output, which can be varied in the sponge construction.

[1] The Merkle-Damgård construction is a method for building cryptographic hash functions. It processes an input message in fixed-sized blocks, padding as necessary, and then compresses each block into a fixed-length hash. This approach, used in SHA-1 and SHA-2, has some vulnerabilities which led to the development of alternatives like SHA-3's sponge construction.

2.1 Walkthrough of the SHA-3 algorithm

Message Padding: The first stage of the SHA-3 algorithm involves preparing the input data, or message. This is done through a technique known as "padding". The padding used in SHA-3 is called "pad10*1" rule. According to this rule, a '1' bit is appended to the message, followed by a series of '0's, and then another '1' bit is appended. The number of '0's added ensures that the total length of the message (in bits) is a multiple of the bitrate.

$$\text{pad}(m) = m||P10^*1 = \ldots, x_1, x_0 \tag{1}$$

State Initialization: In SHA-3, the state is initialized to a string of zeroes. It's a 5x5 array of 64-bit words, resulting in a b=1600-bit state. This state is divided into two sections: the bitrate (r) and the capacity (c). These parameters are chosen according to the required security level and depending on the desired output length of the hash function. For SHA-3-256, which produces a 256-bit hash, the bitrate (r) is set to 136 bytes, and the capacity (c) is set to 64 bytes. For SHA-3-512, which produces a 512-bit hash, the bitrate (r) is set to 72 bytes, and the capacity (c) is set to 128 bytes

Absorbing Phase: The padded message is divided into n blocks(eq.2), each of the size equal to the bitrate. These blocks are processed one by one. For each block, it is XORed with the corresponding part of the state (the first r bytes). Then, a round of the Keccak-f permutation is applied to the whole state. An hight level view of this pahse is shown in Fig.1

$$\text{n} = \frac{\text{len}(\text{pad}(m))}{r} \tag{2}$$

One round of Keccak-f permutation involves a sequence of operations (the sequence is repeated 24 times for each permutation round in SHA-3):

- **Theta (θ)**: This operation consists of two steps. First, it calculates the parity (even or odd number of '1' bits) of each column in the 5x5 array. Then, each bit in the array is XORed with both the parity of the column to its right and the inverse of the parity of the column to its left (considering the array as a torus, i.e., wrapped around). This provides diffusion across columns
- **Rho (ρ)**: In this step, each bit in the 5x5 array is rotated to the left by a varying number of positions. Each bit position has a distinct rotation

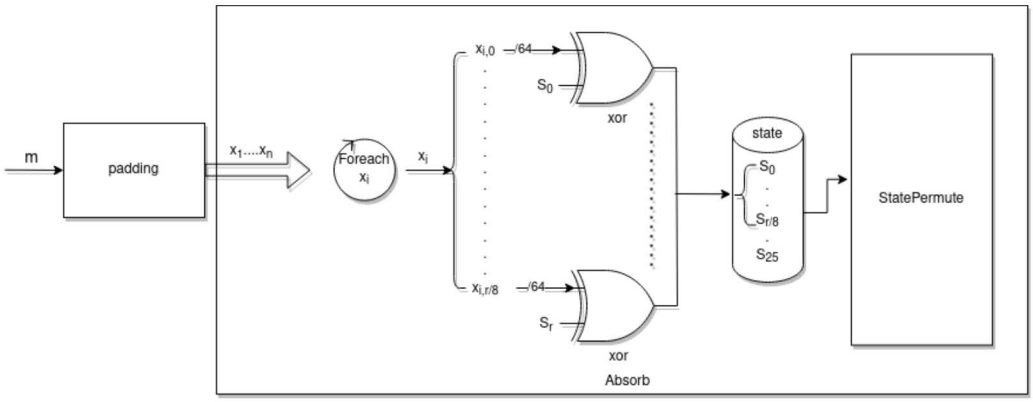

Figure 1: High-level view of Keccak padding an absorb

offset, according to a predefined pattern. This introduces asymmetry into the state.

- **Pi (π)**: This operation simply permutes the bit positions within the 5x5 array, providing additional diffusion across lanes (rows and columns). Each lane of bits is shifted by a certain offset to the left, with the offset varying for each row.

- **Chi (χ)**: This is a non-linear operation applied to each row of the array. Each bit is XORed with the AND of the next bit and the negation of the bit after that (considering wrapping around within the row). This operation doesn't provide diffusion but contributes to the non-linearity of the transformation.

- **Iota (ι)**: In this final step, a predefined round constant is XORed into the array. The round constant is different for each of the 24 rounds of operations and only affects the first lane of the array. This step breaks the symmetry and introduces complexity into the computation.

Squeezing Phase: After the absorption phase in SHA-3, the squeezing phase begins. This phase extracts the hash result from the state. During the squeezing phase, the bits from the state are read out as the output hash, this phase applies the Keccak-f permutation at least once more before extracting the final hash value. The number of output bits produced in each round of the squeezing phase is equal to the size of the bitrate (r).

If the required hash length is larger than the bitrate (r), the squeezing phase doesn't end after one round. Instead, the Keccak-f permutation function is applied to the state again before more output bits are read out. This

continues until enough bits have been squeezed out to form the complete hash.

Important remark

For an exhaustive and detailed explanation of the SHA-3 algorithm, I encourage the readers to refer to the official NIST FIPS 202 standard documentation [1]. Delving into the full details of the SHA-3 operation exceeds the scope of this report. My intention here was to provide a high-level understanding of its working principles

3 Understanding the Nature of DPA Attacks

3.1 Side Channel Attacks

A Side-Channel Attack (SCA) is an attack that targets the information of a cryptographic device, typically the secret key of a cryptographic algorithm. This attack is executed by observing the physical outputs of a device, such as power consumption, operational time, and the emission of heat, light, and sound. The premise behind this approach is that these physical outputs show a correlation with the device's internal state during cryptographic operations. Figure 2 depicts an abstraction of this concept Instead of trying to compromise the primary implementation of a cryptographic device, SCAs focus on monitoring 'leaked' information generated by a device during its regular functioning to infer the secret key leakage. Common instances of such attacks include timing attacks, fault attacks, and power analysis attacks[9].

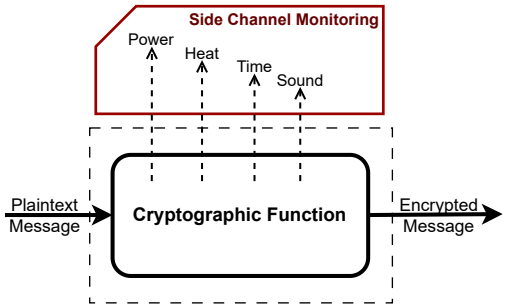

Figure 2: Side channel monitoring

3.2 DPA Attacks and Adversarial model

In a typical DPA attack, an attacker measures the power consumption of a cryptographic device during encryption or decryption processes. These measurements are correlated with the device's computational activities, which directly depend on the key-dependent intermediate results. The attacker then statistically analyzes these measurements to identify patterns or differentials that could be linked back to the secret key. Unlike simple power analysis (SPA) attacks, DPA does not rely on visibly identifiable events in the power consumption. Instead, it relies on statistical techniques applied to a large number of measurements, making it a powerful tool against cryptographic systems.[3]

I used d probing as adversarial model which can be definened as follows:

- The adversary can probe up to 'd' wires within a particular time window.
- When a wire 'g' computing a function 'G' is probed, the adversary can deduce all inputs to 'G' up to the latest synchronization point. This includes all intermediate values during the computation of 'G' and the output of 'G'.
- Refering to Figure 3, if a function 'G' is computed and its output is stored in 'reg3', an adversary probing 'g' can observe all the inputs to 'G', up to and including 'reg2'. They can also calculate all the intermediate values used during 'G', but they cannot directly observe the values stored in 'reg1' or any intermediate values during the computations of each 'Fi'.

This model aligns well with DPA attacks of order 'd', which conside intermediate variables' leakage. Some adversarial models might allow adaptive moving of probes between time periods. However, these are stronger models and this model does not consider this possibility[7]

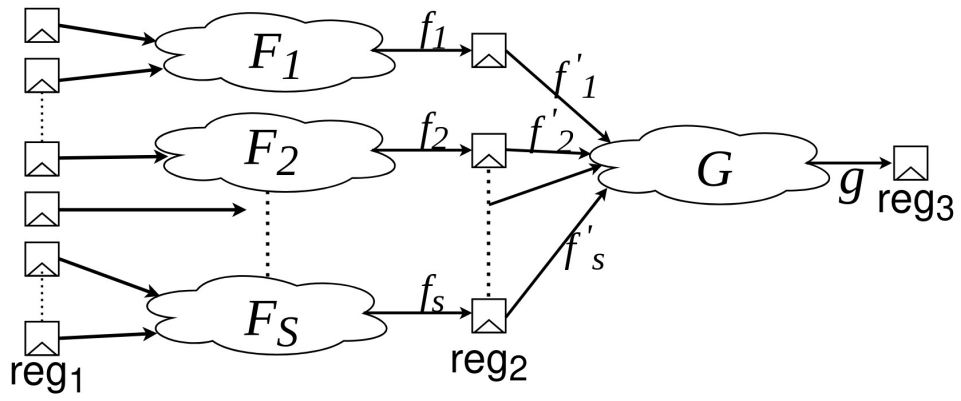

Figure 3: Adversarial model[7]

3.3 DPA and MAC Keccak

When examining the role of a cryptographic function such as Keccak in the context of Message Authentication Codes (MACs), it's important to understand the process involved. Typically, a MAC will concatenate a secret key and the message, or use an XOR operation to combine the key with the message. Following these steps, the combined data is passed through the Keccak hash function. This function assists in verifying that the message originated from the asserted sender, thereby affirming its authenticity, and also ensures that the message has not been altered during transmission, thereby ensuring its integrity. However, even with these robust measures, the security of a MAC is not infallible. For instance, since SHA3 is vulnerable from DPA[4] an attacker could employ this technique and could potentially leak information about the secret key, compromising the overall security.

4 Securing gates

4.1 Preliminaries

In order to introduce Iahai Sahai Wagner masking scheme we need to discuss a cryptographic technique known as **Boolean masking** which is employed to ensure the secure handling of sensitive data during computations. The methodology involves fragmenting a data element '**a**' into 's' distinct parts, referred to as shares $(a_1, a_2, \ldots, a_s)$. This process, known as "s-share representation", maintains the integrity of the data by selecting the first '$s-1$' shares randomly while computing the last share a_s such that $\mathbf{a}=\bigoplus_{i=1}^{s} a_i$, thus

ensuring the accurate reconstruction of the original data. The technique extends to computations involving transformations. For instance, a function $F(a)$ that modifies 'a' into 'b' necessitates the existence of corresponding functions $F_1, F_2, \ldots, F_s$ which operate on each individual share a_i, yielding transformed shares b_i. The new set of shares, collectively $\mathbf{b} = b_1, b_2, \ldots, b_s$, represents the transformed data '$\mathbf{b}$' while preserving the confidentiality of the original data.[6]

4.2 Ishai-Sahai-Wagner masking scheme

The Ishai-Sahai-Wagner (ISW) masking scheme is a procedure for computation on masked data, aiming to protect sensitive data from unwanted probing This technique is a part of a larger group of techniques known as secure multi-party computation (SMPC), where computations are performed on data while keeping the data secret[2][6].

- **Sharing Inputs:** Given the input values **a** and **b**, we want to compute $\mathbf{c} = F(a, b) = \mathbf{a}$ AND **b**. To ensure security against d probes, the ISW scheme takes s = 2d + 1 shares of each input (i.e., the inputs **a** and **b** are each divided into s shares)[7].

- **Randomness Generation:** The ISW scheme consumes $\binom{s}{2}$ bits of randomness. These random bits are represented as z_{ij} where $1 \leq i < j \leq s$ and drawn independently and identically (i.i.d.) from a uniform distribution.

- **Intermediate Computations:** The intermediate values zji are computed as follows: Each z_{ji} is the XOR of three terms: an existing z_{ij} and two terms representing the AND of different shares from **a** and **b**.

- **Output Computation:** The output shares c_i of the result **c** are computed by XORing the product of the corresponding shares of **a** and **b** with relevant intermediate values z_{ij}.

Please note that although the example shows the computation of an AND gate, the ISW scheme can be generalized for other types of gates. This generalization is crucial for implementing a broad range of algorithms in a masked manner, protecting the computation against side-channel attacks.

In the context of secure computations using the ISW masking scheme, SHA-3 can be implemented using only AND and XOR gates. This simplifies the masking process because we only need secure implementations of these

two gates.[2] The generalazed algorithms to compute these gates are reported in algoritms 1 and 2.

Result: s-shares **c** satisfying $c = a \oplus b$
Data: s-shares **a** and **b**

for $i = 1$ **to** s **do**
  $c_i \leftarrow a_i \oplus b_i$
end

Algorithm 1: Algorithm to compute s-shares $\mathbf{a} \oplus \mathbf{b}$

Result: s-shares **c** satisfying $c = ab$
Data: s-shares **a** and **b**

for $i = 1$ **to** s **do**
  for $j = i + 1$ **to** s **do**
    $z_{ij} \leftarrow rnd()$
    $z_{ji} \leftarrow (z_{ij} \oplus a_i b_j) \oplus a_j b_i$
  end
end
for $i = 1$ **to** s **do**
  $c_i \leftarrow a_i b_i$
  for $j = 1$ **to** s, $j \neq i$ **do**
    $c_i \leftarrow c_i \oplus z_{ij}$
  end
end

Algorithm 2: Algorithm to compute s-shares **ab**

Focusing on the complexity in terms of the number of basic operations performed:

- **XOR Gate:** In the context of secure computations using the ISW masking scheme, the computational complexity for an XOR operation is $O(s)$, where s is the number of shares. This is because, Since XOR is a linear function each share of the inputs needs to be XORed only with the corresponding share of the other input[5].
- **AND Gate:** The AND operation is more computationally intensive. According to the ISW masking scheme, the computational complexity for an AND operation is $O(s^2)$, where s is the number of shares. This is because each pair of shares (one from each input) needs to be ANDed, which results in $s * (s - 1)/2$ AND operations. In addition, there are $s * (s - 1)$ XOR operations for calculating the intermediates z_{ji} and another s XOR operations for calculating the output shares c_i, resulting in total $s^2 - s$ XOR operations. So, both the AND and XOR operations contribute to the $O(s^2)$ computational complexity for an AND gate.

[2]Note that in case of first-order DPA attacks, we can use the ISW masking scheme only for the AND gates. XOR gates are naturally resistant to first order because the Hamming weight (number of 1s in binary representation) of the inputs and outputs of an XOR operation is statistically independent. As a result, in this case, we do not need to use a masking scheme for XOR gates.[5]

These complexities become significant when considering higher-order protections (i.e., larger values of s) and large-scale computations such as the ones in cryptographic algorithms like SHA-3. It's also worth noting that the generation of randomness required for the ISW masking scheme, particularly for the AND operation, adds to the overall computational load.

Anyway the overall overhead of implementing the ISW masking scheme on SHA-3 would not be quadratic, despite the quadratic complexity of the AND operations in the ISW masking scheme. This is because the Keccak sponge construction, which is the underlying algorithm of SHA-3, predominantly uses XOR operations and comparatively fewer AND operations.

4.3 Merging ISW and Shuffling

The idea behind shuffling is to randomize the order in which operations are performed to prevent an attacker from correlating the sequence of operations with the data being processed. This makes it harder for an attacker to gain information from the execution patterns of the algorithm[8].

Here's a brief overview of how shuffling has been implemented in the ISW masking scheme:

- **Randomizing the Order of Shares:** An effective shuffling method involves randomizing the order of shares prior to performing computations. Within the context of the ISW masking scheme, shares are not processed in a static sequence i.e. $(a_1, a_2, \ldots, a_s)$. Instead, a random permutation of shares is created and processed, as depicted in the XOR gate implementation (refer to Algorithm 4). Another application of this is observed during share calculation (see Algorithm 3). In this algorithm, a share, denoted by idx, is randomly selected for XOR operation with the input **a**. This approach introduces randomness in two aspects: the position of shares and the timing of the XOR operation with the original input num.

- **Randomizing the Order of Operations:** In addition to randomizing the order of shares, we can also randomize the order in which we compute the XOR and AND operations. For example, when computing the intermediate values z_{ji} and the output shares c_i in the ISW scheme for the AND gate (refer to Algorithm 5), instead of always following the same sequence of computations, we compute these values in a random order.

Input : a
Output: A = s-shares **a**

Allocate an array containing all integers from 0 to s-1
shuffled [s]
Shuffle the array
`shuffle_array(shuffled)`
idx $\leftarrow$ `Rnd` () mod_{s-1}
A[shuffled *[s-1]*] $\leftarrow$ **0**

for $i = 0$ **to** *s-1* **do**
  A[shuffled *[i]*] $\leftarrow$ `Rnd()`
  A[shuffled *[s-1]*] $\leftarrow$ A[shuffled *[s-1]*] $\oplus$ A[shuffled *[i]*]
  if $i = idx$ **then**
    A[shuffled *[s-1]*] $\leftarrow$ A[shuffled *[s-1]*] $\oplus$ **a**
  end
end

Algorithm 3: Share calculation procedure with shuffling

Result: s-shares **c** satisfying $c = a \oplus b$
Data: s-shares **a** and **b**

shuffled [s]
Shuffle the array
`shuffle_array(shuffled)`
for $i = 1$ **to** s **do**
  j $\leftarrow$ shuffled [i]
  $c_j \leftarrow a_j \oplus b_j$
end

Algorithm 4: Algorithm to compute s-shares **a** $\oplus$ **b** with shuffling

Result: s-shares **c** satisfying $c = ab$
Data: s-shares **a** and **b**

shuffled [s]
Shuffle the array
`shuffle_array(shuffled)`

for $i = 1$ **to** s **do**
  k ← shuffled [i]
  for $j = i + 1$ **to** s **do**
    $z_{kj} \leftarrow rnd()$
    $z_{jk} \leftarrow (z_{kj} \oplus a_k b_j) \oplus a_j b_k$
  end
end

Shuffle the array
`shuffle_array(shuffled)`

for $i = 1$ **to** s **do**
  $c_i \leftarrow a_i b_i$
  for $j = 1$ **to** s **do**
    k ← shuffled [j]
    if $k \neq i$ **then**
      $c_i \leftarrow c_i \oplus z_{ik}$
    end
  end
end

Algorithm 5: Algorithm to compute s-shares **ab** with shuffling

The masking scheme, built around the ISW protocol, provides a robust framework for secure computations[8]. By understanding how to randomize the order of shares and the timing of XOR operations, we've added another layer of security in the computations. With this knowledge, we will noe proceed to the actual implementation of this masking scheme on the SHA-3 algorithm.

5 Implementation

5.1 Securing sha3

Upon establishing the methodology for secure computation of XOR and AND gates, it becomes necessary to modify certain aspects of the original SHA-3 computation to accommodate the secure masking scheme. These adjustments

primarily revolve around three main changes:

- **State Transformation:** The original SHA-3 state, constituted by a 5x5 array of 64-bit words, undergoes a transformation to cater to our masking scheme. It is expanded to a 5x5xs array, where 's' represents the number of shares in our masking scheme. This transformation is crucial as it enables the substitution of the original, unprotected gates with our newly defined, secured gates.

- **Absorbing Phase Adjustment:** The adjustment extends to the absorbing phase, where the padded message is segmented into 'n' blocks and XORed with the corresponding part of the state. As detailed in section 2, during this phase, the message is split into 64-bit words. Subsequently, each of these words is divided into 's' shares before being XORed with the corresponding word of the state. This division ensures that the XOR operation is performed securely, maintaining the integrity of the masking scheme.

- **Final Share Recombination:** The final step in the computation process involves a recombination of shares to reconstruct the actual state. This crucial step is performed exclusively at the **conclusion** of the computation. The approach ensures that the original, unshared data is obtained only at the end while all operations on the state throughout the computation steps are performed using secure gates. Consequently, the consistency and security of the masking scheme are maintained throughout the computation, safeguarding the data from potential security breaches.

By incorporating these alterations into the SHA-3 computation, we are able to effectively execute the masking scheme, thereby strengthening the security of the algorithm against potential DPA attacks.

6 Implementation on STM32F4 Microcontroller

The primary goal of the presented implementation is to integrate the masking scheme, involving secure XOR and AND gates, into the SHA-3 cryptographic algorithm on the STM32F4 microcontroller. I've adapted the original encryption algorithm to accommodate these changes and have used the microcontroller's in-built Hardware Random Number Generator (RNG) to generate the random values necessary for our masking scheme.

Here is a description of the main parts of the program:

- **Hardware Configuration:** The hardware resources of the STM32F4 microcontroller are set up in the initial phase. This includes the system clock configuration to operate at 84MHz, initialization of GPIO and USART for serial communication, and enabling the RNG for generating random numbers.

- **Main Loop:** The program then enters a loop that prints a welcome message, reads the mode of operation from the user (sha3_256, sha3_512, shake_128, shake_256), and handles the encryption logic based on the selected mode. Once the user has chosen the encryption mode, the program enters a continuous loop. This loop allows for the encryption of multiple messages without the need to restart or reconfigure the program each time. Users can continue to input strings for encryption, and the program will process these inputs based on the previously selected encryption logic. The loop will run indefinitely, until the program is manually interrupted or halted. This enables convenient and continuous operation for scenarios that involve the encryption of multiple messages or repeated encryptions.

In the computation phase, the ISW masking scheme is integrated within the SHA-3 algorithm. The function reads the user's input string, divides it into shares, performs the SHA-3 operations using the secured gates, and finally, combines the shares to obtain the final output. The output is then sent back to the user through the USART. During the computation, a pin (GPIOB, GPIO6) is set high before each encryption operation and set low afterwards. This pin could be monitored using an oscilloscope to test for any potential side-channel leakage.

The successful integration of the ISW masking scheme into the SHA-3 computation on this microcontroller demonstrates the practical feasibility and effectiveness of this approach in enhancing cryptographic security.

7 Evaluation and Results

The results presented in the semilog plot 4 offer key insights into the practical implications of applying the ISW masking scheme to the SHA-3 cryptographic algorithm, specifically concerning its computational overhead. The plot shows four logarithmic curves representing the relationship between elapsed time and message length for standard SHA-3 and its counterparts with 3, 5, and 7 shares in the ISW masking scheme.

In the plot, we observe that the time complexity of the operation increases linearly with the length of the message for all scenarios. This is consistent

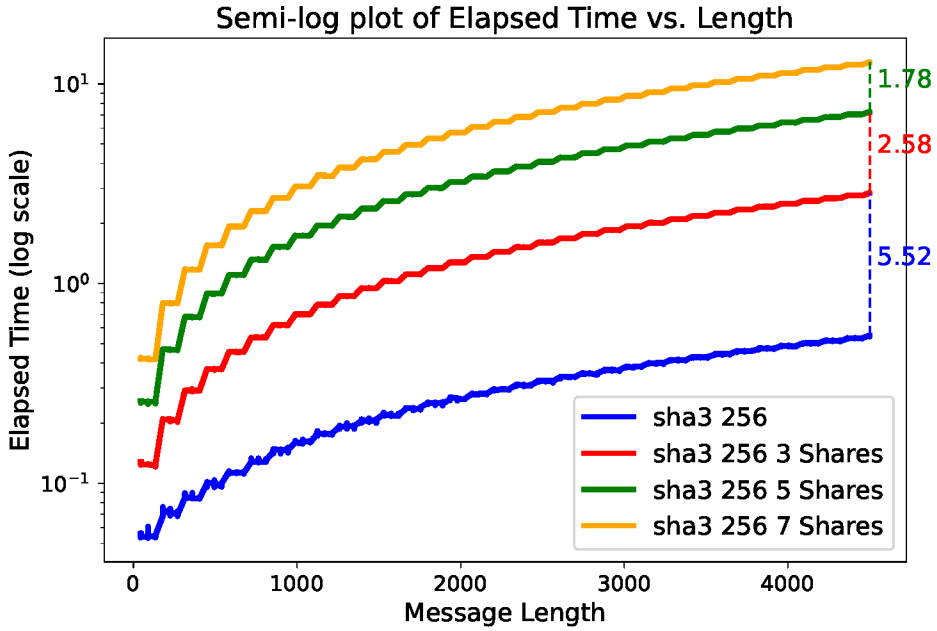

Figure 4: Semi logarithmic plot of elapsed time vs. lenght

with the nature of cryptographic operations where processing larger datasets inherently requires more time. Furthermore, with the increase in the number of shares in the ISW masking scheme, there is a noticeable increase in the elapsed time, which is indicative of the added computational overhead associated with the masking scheme.

Specifically, the multiplicative variations in elapsed time between the standard SHA-3 and the 3, 5, and 7 shares versions are 5.52, 14.22, and 25.31, respectively. Although the masked AND gates in the ISW scheme inherently have a quadratic complexity (s^2, where s is the number of shares), the overall complexity doesn't strictly follow this trend. This can be attributed to the operational profile of the SHA-3 algorithm, particularly the Keccak sponge construction, which involves a significantly larger number of linear XOR operations compared to quadratic AND operations.

Therefore, despite the security enhancement provided by the ISW masking scheme, the cost is an increased computational load. However, due to the preponderance of XOR operations in the Keccak construction, this increase doesn't strictly follow the expected quadratic trend, resulting in a more moderate overall increase in computation time. The choice of the number of shares in the ISW scheme, therefore, represents a critical trade-off between security and computational performance.

References

[1] Morris Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. en. 2015-08-04 2015. DOI: https://doi.org/10.6028/NIST.FIPS.202.

[2] Yuval Ishai, Amit Sahai, and David Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology - CRYPTO 2003*. Ed. by Dan Boneh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 463–481. ISBN: 978-3-540-45146-4.

[3] Owen Lo, William J. Buchanan, and Douglas Carson. "Power analysis attacks on the AES-128 S-box using differential power analysis (DPA) and correlation power analysis (CPA)". In: *Journal of Cyber Security Technology* 1.2 (2017), pp. 88–107. DOI: 10.1080/23742917.2016.1231523. eprint: https://doi.org/10.1080/23742917.2016.1231523. URL: https://doi.org/10.1080/23742917.2016.1231523.

[4] Luo and Pei. *Side-channel security analysis and protection of SHA-3*. 2017.

[5] "Masking". In: *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Boston, MA: Springer US, 2007, pp. 223–244. ISBN: 978-0-387-38162-6. DOI: 10.1007/978-0-387-38162-6_9. URL: https://doi.org/10.1007/978-0-387-38162-6_9.

[6] Lauren De Meyer, Felix Wegener, and Amir Moradi. *A Note on Masking Generic Boolean Functions*. Cryptology ePrint Archive, Paper 2019/1247. https://eprint.iacr.org/2019/1247. 2019. URL: https://eprint.iacr.org/2019/1247.

[7] Oscar Reparaz et al. *Consolidating masking schemes*. Cryptology ePrint Archive, Paper 2015/719. https://eprint.iacr.org/2015/719. 2015. URL: https://eprint.iacr.org/2015/719.

[8] Nicolas Veyrat-Charvillon et al. "Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note". In: *Advances in Cryptology – ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 740–757. ISBN: 978-3-642-34961-4.

[9] Yongbin Zhou and DengGuo Feng. "Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing." In: *IACR Cryptology ePrint Archive* 2005 (Jan. 2005), p. 388.